An Enhanced Evolutionary Technique for the Generation of Compact Reconfigurable Scan-Network Tests

An Enhanced Evolutionary Technique for the Generation of Compact Reconfigurable Scan-Network Tests

Submitted for the Special Issue on Design, Technology, and Test of Integrated Circuits and Systems

Riccardo Cantoro, Aleksa Damljanovic, Matteo Sonza Reorda, Giovanni Squillero

Dipartimento di Automatica e Informatica, Politecnico di Torino,
Corso Duca degli Abruzzi 24, Turin, 10129, Italy
{*riccardo.cantoro, aleksa.damljanovic, matteo.sonzareorda, giovanni.squillero*}*@polito.it*

Nowadays many Integrated Systems embed auxiliary on-chip instruments whose function is to perform test, debug, calibration, configuration, etc. The growing complexity and the increasing number of these instruments have led to new solutions for their access and control, such as the IEEE 1687 standard. The standard introduces an infrastructure composed of scan chains incorporating configurable elements for accessing the instruments in a flexible manner. Such an infrastructure is known as Reconfigurable Scan Network or RSN. Since permanent faults affecting the circuitry can cause malfunction, i.e., inappropriate behaviour, detecting them is of utmost importance. This paper addresses the issue of generating effective sequences for testing the reconfigurable elements within RSNs using evolutionary computation. Test configurations are extracted with automatic test pattern generation (ATPG) and used to guide the evolution. Post-processing techniques are proposed to improve the evolutionary fittest solution. Results on a standard set of benchmark networks show up to 27% reduced test time with respect to test generation based on RSN exploration.

Keywords: IEEE 1687; scan-chain; genetic algorithms, microprocessor testing, ATPG.

1. Introduction

Almost all modern Integrated Circuits (ICs) include resources for supporting test, such as Built-In Self-Test (BIST) modules, sensors for monitoring internal parameters, like current, temperature or delays, and registers for setting up and calibrating the behavior of specific modules, like analog ones.

To unify and simplify the access to all these resources, IEEE published the IEEE 1687 standard [1], which extends and complements the previous IEEE 1149.1. The new standard specifies how to split a scan chain accessible through the JTAG's Test Access Port (TAP) and to program its configuration, allowing the designer to flexibly trade-off between area, access time and other parameters. The resulting *Reconfigurable Scan Networks* (RSNs) support the serial access to the Test Data Registers (TDRs) associated to internal instruments. Indeed, the newest version of the IEEE 1149.1 standard [2] also includes ways to design similar scan networks.

In IEEE 1687, typical RSNs are chains of flip flops interleaved with *Segment Insertion Bits* (SIBs) and *ScanMuxes* (SMs) that allow to dynamically split the whole scan chain into segments that may be connected in series or in parallel. Using RSNs, a faster and more efficient access to the resources is possible: the user first configures the network, selecting the subset of instruments to be accessed, then uses the network to serially read and write the required data. CAD tools already have support for the automating introduction of RSNs [3].

Several works focused on the test of possible permanent faults affecting a standard scan chain, e.g., by shifting into the chain a sequence of alternated 0s and 1s, and checking that the same sequence appears at the other extreme of the chain [4–6]. Indeed, testing an RSN is far more complex, as it is necessary also to check whether the network can be properly configured and whether it works as expected after the configuration (i.e., whether the expected sub-network is made accessible), whichever legal configuration we enforce. In particular, this test requires checking whether each special module inserted to support the network configuration works properly.

While some works (e.g., [7]) already faced the issue of testing the test circuitry mandated by the IEEE 1149.1, when adopting a IEEE 1687 RSN, one must also consider the issue of testing the related hardware, and check for possible defects affecting it.

In [8] we proposed a general approach to automatically generate a test sequence for an IEEE 1687 RSN with respect to permanent faults. We provided techniques for testing SIBs and ScanMuxes, and then we described how to combine them into a single comprehensive test. This test is independent on the specific implementation of the network elements and does not require any change in the hardware implementing the network itself. Test generation can directly start from the network structure described using the Instrument Connectivity Language (ICL), as mandated by the IEEE 1687. The proposed test generation algorithms were based on different heuristics that could easily run even on relatively large RSNs.

In [9] we refined that approach to minimize the duration of the resulting test sequence: the faced problem was properly modeled according to the graph theory, and an optimal algorithm able to generate the minimum-duration test sequence was described. Unfortunately, such an approach did only work on relatively small RSNs, and sub-optimal solutions must be accepted when dealing with real cases.

In this paper we make one step forward, and we propose a method that produces test sequences far more effective than the the heuristic solutions proposed in [8], yet that is able to deal with large and complex RSNs. The proposed method always produces a test sequence able to detect all permanent fault affecting the reconfigurable modules, and allows to easily trade-off between the computational effort and the quality of the result.

Experimental results are reported on the set of benchmark networks proposed in [10], which practically demonstrate the effectiveness of the proposed approach.

The paper is organized as follows. In Section 2 we summarize the key characteristics of the IEEE 1687 networks. In Section 3 we propose the techniques for

generating an optimized test sequence for a RSN. Section 4 reports some experimental results, and Section 5 finally draws some conclusions.

2. Background

In this Section we will first briefly overview the key characteristics of an IEEE 1687 RSN, then we will explain how their test is performed in [8], and finally we will summarize why minimizing the test duration may turn into a computationally complex task.

2.1. *Overview of RSNs*

As mentioned in Section 1, a key feature of RSNs is the possibility to partition the set of instruments into segments controlled by programmable components, and then dynamically decide which segments are currently accessible and which are bypassed. The first programmable component introduced by IEEE 1687 is the SIB, which allows to bypass a segment of a network. As a segment can be simply one or several TDRs or a sub-network consisting of TDRs and other programmable components, it is possible to create a hierarchical network.

Fig. 1(left) shows the simplified schematic of a possible implementation of a SIB, which is based on a one-bit shift-update register and a two-input multiplexer. SIBs can be programmed by shifting a bit into their S flip-flop and latching that bit into the parallel U latch. If the latched bit is 0, the SIB is *de-asserted*, and the scan-path is from the *si* terminal, to the *so* terminal via the S flip-flop, bypassing the segment between the tsi and fso terminals. If, on the other hand, the latched bit is a 1, the SIB is *asserted*, and the scan-path includes the segment connected between the tsi and fso terminals of the SIB. In this paper, the symbol shown in Fig. 1(right) is used to represent a SIB.

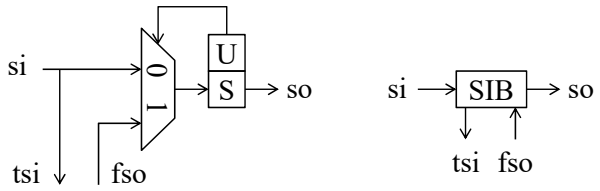

Fig. 1. Simplified schematic of a SIB module (left) and its symbol (right)

IEEE 1687 RSNs can be also constructed using shift-update registers and ScanMuxes. As an example, consider the network shown in Fig. 2(left) in which a two-bit shift-update register is used to select among four inputs of a 4-to-1 ScanMux. Here again, the configuration of the ScanMux can be performed by shifting the required

values into the S shift flip-flops of the control register and then latching the shifted bits into the U latches. In the rest of this paper, the symbol shown in Fig. 2(right) will be used to represent the shift-update register that controls a ScanMux.

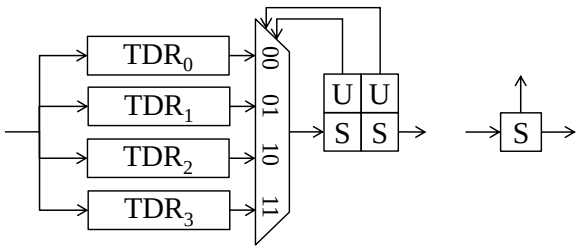

Fig. 2. Simplified schematic of a ScanMux module (left) and its control register's symbol (right)

To keep the drawings simple in Figures 1 and 2, the clock, reset, control signals (namely, *shift*, *update*, and *capture*), and the *select* signal used to gate the control signals are not shown. To follow the examples in this work, it should suffice to assume that only the TDR connected to the selected port of a ScanMux receives (i.e., reacts to) the clock and control signals. It should be noted that the configuration of the network (i.e., the status of the latched bits) does not change when shifting a new vector through the shift cells, but only in the update phase where the shifted vector is latched into the U cells.

To operate an IEEE 1687 network from outside the chip, the IEEE 1149.1 TAP can be used. The TAP finite state machine provides the control signals needed to configure the IEEE 1687 network and access the instruments through it.

As an example, let us consider an RSN that includes five instruments: the user can access them through the TAP port, reading or writing from/to the associated Test Data Registers (TDR_1 to TDR_5). In order to save time when accessing to the instruments, the designer, instead of connecting all TDRs into a single chain, like in 1149.1-complaint circuits, may decide to adopt an IEEE 1687 network including three SIBs and one ScanMux (SM), as shown in Fig. 3; each of these four configuration modules can be configured to allow the access to a given subset of TDRs (and the associated instruments). Table 1 reports sixteen possible configurations supported by this network, which depend on how the SIBs and the ScanMux have been configured. In Table 1, "A" means asserted, "D" means de-asserted, 0 and 1 correspond to the two possible positions of the ScanMux, and "-" appears when a module belongs to an inaccessible segment.

In order to move the network to a given configuration, the user must first shift-in a suitable sequence of bits, so that the S flip-flops of SIBs and ScanMuxes hold the correct value, then activate the update signal to move these bits to the U latches. The sequence of bits to configure the network is called *configuration vector*. A generic configuration vector is referred to as cv_i. Once a configuration is reached,

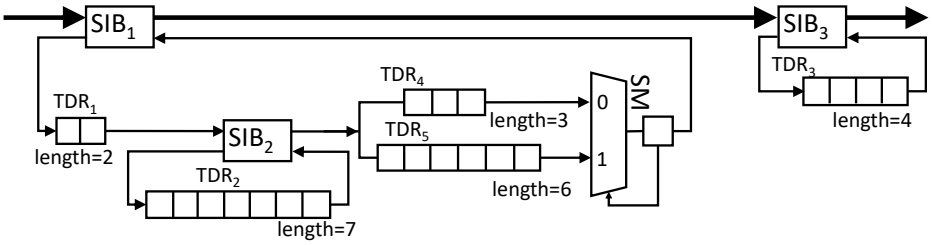

Fig. 3. Example of IEEE 1687 RSN.

Table 1. Set of possible configurations of the RSN in Fig. 3.

Configuration	SIB_1	SIB_2	SM	SIB_3	Active path	Length
C_0	D	D	0	D	-	2
C_1			1			
C_4		A	0			
C_5			1			
C_2	D	D	0	A	TDR_3	6
C_3			1			
C_6		A	0			
C_7			1			
C_8	A	D	0	D	TDR_1, TDR_4	9
C_9	A	D	1	D	TDR_1, TDR_5	12
C_{10}	A	D	0	A	TDR_1, TDR_3, TDR_4	13
C_{11}	A	D	1	A	TDR_1, TDR_3, TDR_5	16
C_{12}	A	A	0	D	TDR_1, TDR_2, TDR_4	16
C_{13}	A	A	1	D	TDR_1, TDR_2, TDR_5	19
C_{14}	A	A	0	A	TDR_1, TDR_2, TDR_3, TDR_4	20
C_{15}	A	A	1	A	TDR_1, TDR_2, TDR_3, TDR_5	23

a given subset of the TDRs is accessible, which constitutes the so-called *active path*. The rightmost column of Table 1 reports the length of the active path for each configuration, which corresponds to the number of TDR and S flip-flops in the path. The reader should note that moving from one configuration to another may require more than one configuration vector. For example, in the network of

Fig. 3 moving from C_2 to C_{13} requires first turning SIB_1 into the asserted state (e.g., moving to C_8) and only then we will be able to change the configuration of the SM scan multiplexer to select input branch 1 and turn SIB_2 into the asserted state. Hence, moving from C_2 to C_{13} requires 2 configuration vectors.

2.2. *Test of RSNs*

Testing a non-recongurable scan chain for permanent faults can be performed by shifting a suitable sequence of 0s and 1s through the scan chain. RSNs are however far more complicated to test: in addition to flip-flops composing the TDRs, which have to be tested to check whether they can correctly shift values when included in the active path, the recongurable modules (i.e., SIBs and ScanMuxes) have to be also tested to check whether they are able to move the network to the corresponding congurations.

In this work we use the high-level fault model which was introduced in [8]. The faults affecting the recongurable modules, such as ScanMuxes, are modeled such that a different conguration is selected rather than the expected one. Such a fault leads to a different active path (called *faulty path*) than the expected one, and the two are likely to have a different length. For example, in Fig. 3 the multiplexer (ScanMux) may be affected by a permanent fault whose effect is that the segment connected to the input 0 is always selected, no matter the value in the selection cell. The same may arise for the generic SIB_i, which can be affected by faults named *stuck-at asserted* (SIB_i-s@A) and *stuck-at de-asserted* (SIB_i-s@D). The stuck-at faults in the scan bits of the selection cells are considered as detected by implication by testing such high-level faults, which cover also the faults affecting the update logic of the recongurable modules.

Moreover, such faults cover some faults affecting the reset logic, whose effect is that the module is stuck at the reset value. The other reset faults (i.e., those that make the reset ineffective) are not considered but can be targeted by employing the techniques described in [11].

Resorting to these high-level fault models, one can test an RSN by first configuring the RSN so that the target fault is excited, and then comparing the length of the activated path against the length of the expected path. As an example, the high-level fault that affects the ScanMux of Fig. 3, to always select the segment connected to the input 1, can be excited by a conguration which selects the input 0; congurations C_{12} and C_{14} fulfill this requirement. Once one of them is activated, one can measure the length of the active path by shifting a given sequence (called *test vector*) in TDI and checking when it will appear on TDO. Any fault modifying the length of the active path can be detected in this way. A generic test vector is referred to as tv_i in this paper.

In order to test all configurable modules in a RSN, we can thus organize the test sequence in *sessions*: in each session we first configure the network via one or more configuration vectors (so that each SIB and each ScanMux is switched into a given

position), and then check whether the expected path has been inserted between TDI and TDO via a test vector, i.e., whether the right segments can be accessed. Since the number of possible configurations of a network grows exponentially with the number of configurable modules, the problem of identifying a sequence of sessions which guarantees that 1) all the configurations modules are fully tested, and 2) the total test duration is minimized, is not trivial. Coming back to the example of Fig. 3, this means identifying the sequence of configurations (out of the 16 possible ones) that matches the two above goals.

This paper describes a new method to automatically generate at least one test sequence able to test all configurable modules with minimal duration for any RSN.

3. Methodology

The approach proposed in this paper aims at generating an effective test sequence able to detect all testable faults while requiring a reduced test application time. In a first phase, an evolutionary algorithm [12] is used to cultivate a population of individuals representing a set of RSN configurations in which test vectors are applied. Then, the best individual produced in terms of test application time is further optimized by a post-processing algorithm, which tries to anticipate some test vectors and to remove redundant configurations.

In the following, the basic concepts needed to understand the proposed approach are briefly introduced (Section 3.1). Details are then given concerning: the algorithm used to perform a transition from a given configuration to a target one (Section 3.2), the evolutionary algorithm (Section 3.3), the encoding used for defining individuals (Section 3.4), and the post-processing algorithm (Section 3.5).

3.1. *Basics*

The proposed approach requires the following features:

(1) A function (referred to as $Transition$) able to produce a sequence of configuration vectors cv_1, cv_2, ..., cv_n that moves the RSN from the generic configuration C_{src} to the configuration C_{dst}. The configuration vectors cv_1, cv_2, ..., cv_n are applied (i.e., shifted in the network through scan input pins for as many clock cycles as the active path length, and followed by an update operation), the first (i.e., cv_1) starting from C_{src} and passing through several intermediate configurations (i.e., C_1, C_2, ..., C_{n-1}) up to C_{dst}. This function can be associated to a cost in terms of clock cycles required to apply all the configuration vectors generated.

(2) A function $Evaluation$ able to produce the list of faults that can be excited when the RSN is moved to the generic configuration C_i. Such faults would be covered by means of a test vector applied after reaching C_i. This function can be applied to a set of configurations; in such a case, $Evaluation(C_1, C_2, ..., C_n)$ produces the list of faults covered by all the configurations in the set: if a test

vector tv_i is applied in each of the evaluated C_i, then all faults are covered.

By using an evolutionary engine which calls the *Transition* and *Evaluation* functions we aim at identifying a sequence of configurations detecting all faults and having minimum cost. Each configuration is associated to a test session. Each test session is composed by applying the *Transition* function to generate intermediate configuration vectors which move the network from the configuration C_i in the list to C_{i+1}. The first time, the function is applied between the reset configuration C_{rst} and the first configuration in the list (if not equal to C_{rst}). A test vector is applied to the RSN after each transition to a configuration in the list. Thus, the *Evaluation* function is applied to the list of configurations and the faults obtained are used to compute the fault coverage. Moreover, the total test time is obtained as the cost to apply the configuration vectors generated by the *Transition* functions plus the time required to shift all test vectors.

As an example, let us consider the RSN in Fig. 3. For this network, let us suppose the reset configuration is the one indicated with C_0 in Table 1. A possible solution to the problem of testing the network faults consists in the following sequence of configurations: $[C_0, C_8, C_{15}]$. For each configuration C_i a test vector tv_i is applied, which is made as follows:

(1) as many 0s as the longest path length, i.e., 23 bits in the example network;
(2) an alternated sequence 0101..., as long as the length of the active path currently selected;
(3) two consecutive 1s (or two consecutive 0s) as the sequence terminator;
(4) only for the last test vector, a sequence as long as the length of the active path currently selected (values being shifted in are not important).

As highlighted in the list of faults excited by each configuration (see Table 2), the configurations $[C_0, C_8, C_{15}]$ allow detecting all faults in the network. The list of vectors corresponding to such list of configurations is composed as follows:

(1) tv_1 in C_0 (shift of 23+2+2 bits)
(2) cv_1 from C_0 to C_8 (shift of 2 bits, then update)
(3) tv_2 in C_8 (shift of 23+9+2 bits)
(4) cv_2 from C_8 to C_{15} (shift of 9 bits, then update)
(5) tv_3 in C_{15}(shift of 23+23+2+23 bits)

If a cost of 5 clock cycles is considered to move the TAP controller from shift to update and vice-versa (also including the first shift after the network reset), the above test sequence is executed in 168 clock cycles.

The order in which configurations appear in the list is important and results in different vectors generated by the *Transition* function. For example, let us consider the same set of configurations as in the previous example but listed in a different order: $[C_0, C_{15}, C_8]$. In this case, the list of vectors composing the test sequence is the following:

Table 2. List of faults excited by the RSN in Fig. 3.

Configuration	Set of covered faults
C_0 C_1 C_4 C_5	SIB_1-s@A, SIB_3-s@A
C_2 C_3 C_6 C_7	SIB_1-s@A, SIB_3-s@D
C_8	SIB_1-s@D, SIB_2-s@A, SM_1-s@1, SIB_3-s@A
C_9	SIB_1-s@D, SIB_2-s@A, SM_1-s@0, SIB_3-s@A
C_{10}	SIB_1-s@D, SIB_2-s@A, SM_1-s@1, SIB_3-s@D
C_{11}	SIB_1-s@D, SIB_2-s@A, SM_1-s@0, SIB_3-s@D
C_{12}	SIB_1-s@D, SIB_2-s@D, SM_1-s@1, SIB_3-s@A
C_{13}	SIB_1-s@D, SIB_2-s@D, SM_1-s@0, SIB_3-s@A
C_{14}	SIB_1-s@D, SIB_2-s@D, SM_1-s@1, SIB_3-s@D
C_{15}	SIB_1-s@D, SIB_2-s@D, SM_1-s@0, SIB_3-s@D

(1) tv_1 in C_0 (shift of 23+2+2 bits)
(2) cv_1 from C_0 to C_8 (shift of 2 bits, then update)
(3) cv_2 from C_8 to C_{15} (shift of 9 bits, then update)
(4) tv_2 in C_{15}(shift of 23+23+2 bits)
(5) cv_3 from C_{15} to C_8 (shift of 23 bits, then update)
(6) tv_3 in C_8 (shift of 23+9+2+9 bits).

The above test sequence has the same fault coverage of the previous example but is longer to execute (182 clock cycles). Moreover, it can be noticed that the configuration C_8 is visited twice before applying a test vector (tv_3).

3.2. *Transition function*

The TRANSITION function computes the sequence of configuration vectors able to move the network state from the starting configuration to a target one with minimal configuration cost (Algorithm 1).

The CONFIGUREBRANCH function composes the portion of the next state that is required to configure each multiplexer in the current branch towards the tar-

Algorithm 1 Transition function

```
function TRANSITION(C_src, C_dst)
    p ← ( )                                              ▷ Empty sequence of inputs
    C_next ← C_src
    hasNext ← true
    while next do
        hasNext ← CONFIGUREBRANCH(C_src, C_next, C_dst, 0, confBits)
        if hasNext then
            Append C_next to p
    return p
```

get state (Algorithm 2). First, the total number of steps required to configure a multiplexer is calculated taking into account the target state of sub-hierarchical multiplexers it controls. Then, the maximum number of steps for all multiplexers in a given branch is set as a number of steps required to configure that branch. Subsequently, all multiplexers that require the highest number of configuration steps are immediately configured to match the target state. Conversely, the ones that do not, are configured to match the minimal possible length configuration starting from the higher hierarchical levels so that the previously calculated number of required configurations is not affected.

Algorithm 2 Configuring the branch for next configuration

```
function CONFIGUREBRANCH(C_src, C_next, C_dst, start, end)
    branchSteps ← BRANCHCONFIGSTEPS          ▷ num. of steps for conf. branch
    i ← start                                              ▷ scanning the branch
    while i < end do
        configureBranch
        Mux ← {multiplexer controlled by i configuration bit}
        if Mux not accessible then
            continue
        muxSteps ← MUXCONFIGSTEPS                ▷ num. of steps for conf. mux
        if branchSteps > 1 and muxSteps < branchSteps then
            hasNext|= MINIMIZEMUX(C_src, C_next, C_dst, Mux)   ▷ some state var.
on mux branches need to be conf.
        else if muxSteps = branchSteps then
            hasNext|= CONFIGUREMUX(C_src, C_next, C_dst, Mux)
        i ← next top level multiplexer
    return hasNext
```

The function CONFIGUREMUX composes the portion of next state needed to configure the given multiplexer toward the target state. First, it recursively com-

poses the next state configuration for the selected branch of the multiplexer. Then it composes the next state that the multiplexer itself must assume, selecting the shortest branch that still needs to be configured. The function returns true if some of the state variables in the multiplexer branches still need to be configured, false otherwise. When multiple branches of the multiplexer have to be configured, these are configured in the order of their current scan path length in order to minimize the cost of switching between these branches.

The function MINIMIZEMUX composes the portion of the next state needed to configure the given multiplexer toward its minimal length configuration. However, often imposing the minimum length configuration on the multiplexer may result in changing (increasing) the maximum number of steps required to reach the target state. Therefore, first it calculates which branch should be configured to minimize the multiplexer length. Then it configures the multiplexer and its branches to match the target configuration. If the new selection of the branch corresponds to the minimum length branch, the length of that branch is minimized. Otherwise, the branch with the minimum length is selected. The function returns true if some of the state variables on the multiplexer branches still need be configured, false otherwise.

3.3. *Evolutionary algorithm*

The proposed approach exploits an evolutionary meta-heuristic to identify a test sequence which minimizes the test cost while guarantying the full test coverage. A population of *individuals* is cultivated by the *evolutionary engine*; each individual corresponds to a variable-length sequence of valid configurations $\{C_{t0}, C_{t1}, C_{t2}, ..., C_{tk-1}\}$.

Individuals are evaluated by a separated *evaluation engine* that provides the evolutionary engine with the *fitness values* of each individual. In more details, the evaluation engine:

(1) applies the $Evaluation$ function to the list of configurations and computes the fault coverage;
(2) generates the test sequence, composed of configuration vectors obtained by applying the $Transition$ function between consecutive configurations in the list, and test vectors $\{C_{t0}, \{C_{0i}\}, C_{t1}, \{C_{1i}\}, C_{t2}, \{C_{2i}\}, ..., C_{tk-1}\}$; then, it computes the cost in terms of time (number of clock cycles) needed to execute the latter sequence.

The evolutionary framework is given in Fig. 4. In the proposed flow, the fitness of an individual is composed of two components: the fault coverage and the inverse of the test cost. These components are considered lexicographically: if the fault coverage is higher, the fitness is higher, independently from the test costs.

At the beginning of the evolution, a population of n_p random individuals is generated. Then, in each step, called *generation*, the population is first expanded,

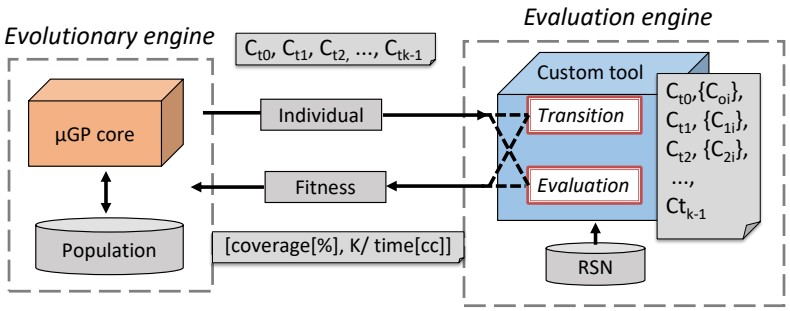

Fig. 4. Evolutionary framework.

then shrunk back to its original size.

During the expansion, n_o genetic operators are activated and the generated offspring is added to the population, in a *steady-state* approach. Genetic operators include the standard mutation operators, that generate a new candidate solution by slightly modifying an existing one, and crossover operators, that generate a new candidate solution by recombining two existing solutions. Then the evaluation engine is used to assess the fitness of all the new individuals. Finally, the least fit individuals are discarded, shrinking the size of the population down to the original n_p.

The process is iterated until a steady state is detected. That is, the fittest individual in the population does not change for a given number of generations. Such a condition intuitively indicates that a local optimum has been reached.

Alternatively, some individuals able to cover all faults can be directly inserted in the initial population. This technique, called *seeding*, is likely to speed up the evolutionary process: the optimizer is only asked to reduce the cost and not to saturate the fault coverage first. However, the offspring of these few initial individuals could take over the entire population quickly, bringing the algorithm into a local optimum. The experimental analyses suggest using seeding only when it is particularly hard to reach the full test coverage of the considered RSN.

3.4. *Individual encoding*

Each individual created by the evolutionary engine consists in a sequence of configurations. Since a configuration is determined by values in the selection bits of each reconfigurable element in the RSN, it can be represented by a bit-string. Individuals are thus files composed of multiple bit-strings.

The evolutionary engine creates individuals which are structured as described in a constraint library. The constraint library is also saved in a file and contains one or more *macros*, each one defining a possible mapping of a bit-string in the individual. In other words, in order for an individual to be considered as valid by

the evolutionary engine, each of its lines must match one of the macros in the constraint library.

In the problem in hand, a macro describes which parts of a bit-sting are fixed to predefined values and others which can be freely modified by the evolutionary engine. As an example, if the RSN in Fig. 3 is considered, a possible macro in the constraint library can be "D−−D", which is satisfied by all configurations in Table 1 having SIB_1 and SIB_3 de-asserted ("−" means don't care). If a macro composed of all don't care bits is included in the constraint library, then the evolutionary engine is allowed to define completely random configurations. Such macro will be referred to as the *random macro*.

In the proposed methodology, other than the random macro, constrained configurations are extracted using automatic test patterns generation (ATPG) on a combinational circuit that represents the problem and converted to macros. The circuit is graphically described in Fig. 5 and receives as input the following values:

(1) as many bits as the number of configuration bits in the RSN (*conf* in the figure);
(2) as many bits as the number of functional faults in the RSN (*faults* in the figure).

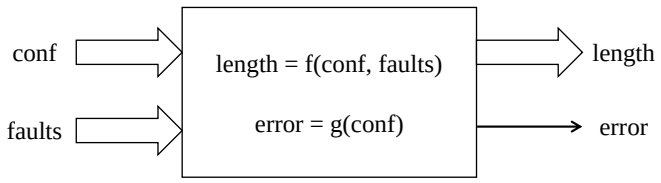

Fig. 5. Combinational circuit used for ATPG.

If one of the input signals of *faults* is set to 1, then the corresponding fault (e.g., SIB_1-stuck-at-asserted) is activated.

As output, the circuit produces the following values:

(1) the active path length (*length* in the figure) in the configuration *conf*, when one or more faults are active (i.e., one or more bits of *faults* are set to 1);
(2) a bit (*error* in the figure) that alerts when an illegal configuration is used as the *conf* value.

The combinational circuit can be written in behavioral VHDL or Verilog by encoding the truth-table of the active path length function (e.g., as in Table 1). However, such an approach becomes easily unfeasible due to a high number of configuration bits or when the RSN is designed using certain patterns (e.g., several sibling SIBs). The approach we suggest is to build the circuit incrementally while traversing the RSN hierarchy. The final length can be expressed as a sum of different contributes associated to TDRs, SIBs, and ScanMuxes. As an example, the final length of the RSN in Fig. 3 is the sum of the lengths associated to the sub-networks

controlled by SIB_1 and SIB_3, respectively. The pseudo-code of the functions LENGTH and ERROR for the example RSN is reported in Algorithm 3.

Algorithm 3 Combinational circuit functions for the RSN in Fig. 3

```
function LENGTH(conf, faults)
    if SIB2 is de-asserted or SIB2-s@D then
        lengthSIB2 ← 1
    else if SIB2 is asserted or SIB2-s@A then
        lengthSIB2 ← 1 + 7
    else
        lengthSIB2 ← 0                                   ▷ unexpected case
    if SM selects 0 or SM-s@0 then
        lengthSM ← 3
    else if SM selects 1 or SM-s@1 then
        lengthSM ← 6
    else
        lengthSM ← 0                                     ▷ unexpected case
    if SIB1 is de-asserted or SIB1-s@D then
        lengthSIB1 ← 1
    else if SIB1 is asserted or SIB1-s@A then
        lengthSIB1 ← 1 + 2 + lengthSIB2 + lengthSM + 1
    else
        lengthSIB1 ← 0                                   ▷ unexpected case
    if SIB3 is de-asserted or SIB3-s@D then
        lengthSIB3 ← 1
    else if SIB3 is asserted or SIB3-s@A then
        lengthSIB3 ← 1 + 4
    else
        lengthSIB3 ← 0                                   ▷ unexpected case
     return lengthSIB1 + lengthSIB3
function ERROR(conf) return 0                            ▷ No illegal configurations
```

In order for the behavioral circuit to be ATPG ready, it is then translated in structural Verilog by means of logic synthesis. The ATPG process consists in the following steps:

(1) in order to activate faults internally, the *faults* input signals are constrained to the value 0;
(2) in order to generate only valid configurations, the *error* output signal is constrained to the value 0;
(3) the ATPG fault list includes stuck-at-1 faults on the *faults* input signals, only;

(4) X values are used as don't care bits in the patterns list.

After performing the ATPG, patterns are saved into a text file and translated into macros and included in the constraint library, such that the evolutionary engine can freely modify don't care bits while fixing the other bits to the values reported in the corresponging pattern.

3.4.1. *Alternative encoding*

A suitable test vector is shifted-in after reaching each configuration. The $Transition$ function interconnects the configurations in the list, eventually adding intermediate configurations where tests are not performed. Therefore, configuration patterns to reach the configuration C_j from C_i are decided by $Transition(C_i, C_j)$, hence also intermediate configurations. Since the purpose of the proposed approach is the minimization of the test time, the $Transition$ function should be able to compute the minimum cost path from C_i to C_j. Alternatively, if a sub-optimal $Transition$ function is available, we propose to slightly modify the structure of the individuals generated by the evolutionary engine.

The alternative encoding consists in adding a flag to each configuration in the list to indicate whether a test vector should be applied in that configuration or not. An example of individual for the RSN of Fig. 3 is $[C_0t, C_8f, C_{12}t, C_{13}t]$, where t indicates that a test vector is applied after reaching that configuration, and f the opposite case. The example can be interpreted as the intention to force the network to pass through the configuration C_8, which becomes an intermediate configuration for the transition between C_0 and C_{12}. In details, it is like splitting $Transition(C_0, C_{12})$ into $Transition(C_0, C_8)$ and $Transition(C_8, C_{12})$. Clearly, the fault coverage is computed by applying the $Evaluation$ function to the configurations that are marked with t, only. This is because faults excited by intermediate configurations are potentially excited but not explicitly observed.

Using the proposed modification, the problem of finding the best path to a configuration that requires a test vector is partially delegated to the evolutionary engine. Clearly, the problem becomes more complex compared to when an optimal $Transition$ function is used; thus, the progression of the evolution becomes slower.

3.5. *Post-processing techniques*

Two post-processing techniques are proposed in order to reduce the test cost of the sequence generated resorting to the evolutionary algorithm described in Section 3.3. They can be applied on the provided test sequence independently, if necessary.

The first one is used to process the full list of configurations in which test is performed and configurations that are exclusively used to interconnect the latter ones. The function reads the list in the reverse order (from end to beginning) and tries to advance the last test vector by appending it next to one of the preceding intermediate transition configurations (Algorithm 4); by doing so, all the configuration

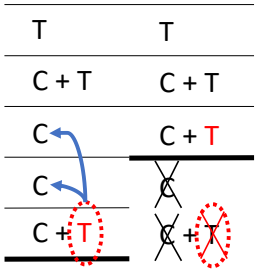

Fig. 6. Post-processing I

Algorithm 4 Bottom-up approach for moving the test states

```
function POSTPROC1(S)
    nextState ← true
    U ← S
    while nextState do
        EVALUATE(U, faultC, costT)        ▷ calculate fault coverage and test cost
        minCost ← costT
        bestSeq ← U
        nextState ← false
        U ← U{ remove last Test vector}
        for s_i ∈ {U} do
            if s_i is Configuration then
                H ← U{insert Test vector at i position}
                H{remove excessive Configuration vectors}
                EVALUATE(H, nfaultC, ncostT)
                if nfaultC = 100% then                    ▷ check coverage
                    if ncostT < minCost then              ▷ check cost
                        minCost ← ncostT      ▷ update cost, save new sequence
                        bestSeq ← H
                        nextState ← true
        U ← bestSeq
    return U
```

vectors required previously to reach the last test state from the penultimate one can be discarded including the last test vector (Fig. 6). Consequently, removing them, the number of clock cycles required to apply the generated test sequence is directly reduced. The condition for advancing such test vector is that the fault coverage has to remain unchanged while the test cost of the modified sequence should be reduced. If the last test vector is successfully anticipated, the algorithm continues

checking the updated test sequence. This operation is performed until it becomes impossible to satisfy the condition and move forward currently last test vector in the modified test sequence.

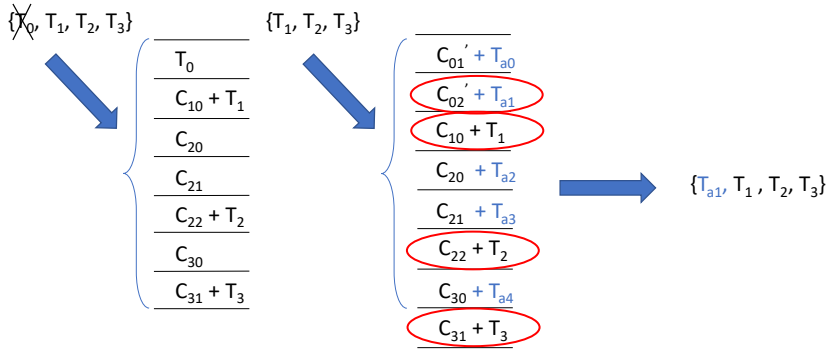

Fig. 7. Post-processing II, first test vector removed

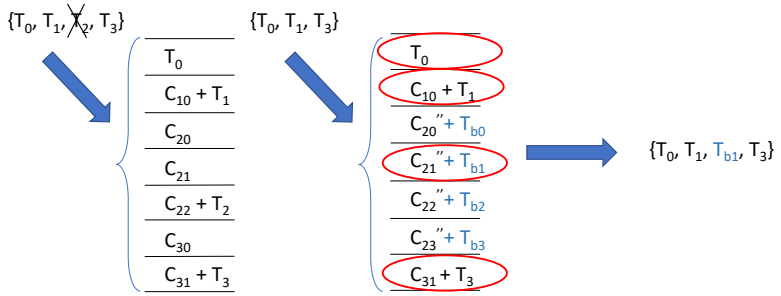

Fig. 8. Post-processing II, third test vector removed

The second technique is used to perform modifications on the test vector set (Algorithm 5). For each test vector in the list, a new (reduced by one) list is generated excluding that particular vector. The new list is generated by applying the *Transition* function on all pairs of consecutive test vectors to insert interconnecting configuration vectors. All of the configuration vectors are considered as potential candidates to be followed by a test vector, based on the set of faults they cover. After traversing the whole list to find potential points of test vector insertion, the newly generated list is evaluated and recorded only if the fault coverage is unchanged (100%) while the test cost is reduced with respect to the one previously

Algorithm 5 Removing test states and trying to insert new ones with reduced cost

```
function POSTPROC2(T)
    CT ← T{apply TRANSITION()}        ▷ interconnect Test vect. with Conf. vect.
    EVALUATE(CT, faultC, costT)        ▷ calculate fault coverage and test cost
    minCost ← costT
    bestSeq ← T
    hasNext ← true
    while hasNext do
        hasNext ← false
        for t_i ∈ {bestSeq} do
            U ← bestSeq{remove t_i test vector }        ▷ remove one Test vector
            FSet ← U{set of faults covered by set of Test vectors}
            TList ← U{apply TRANSITION()}        ▷ add interconnecting Conf.
            NTList ← ( )
            for s_i ∈ TList do
                if s_i is Configuration then        ▷ among Conf. vect. try to
                    if FAULTS(s_i) \ FSet ≠ ∅ then        ▷ insert Test vect. to increase
                        Append s_i to NTList        ▷ fault coverage
                else
                    Append s_i to NTList
            EVALUATE(NTList, nfaultC, ncostT)        ▷ evaluate new Test vect. list
            if nfaultC = 100% then
                if ncostT < minCost then        ▷ save the better solution
                    minCost ← ncostT
                    bestSeq ← NTList
                    hasNext ← true
    return bestSeq
```

recorded. The process is repeated until no further improvement is possible for a given sequence of test vectors. The Fig. 7 and Fig. 8 show the algorithm flow and exemplify how removing different test vectors from the initial list results in having different interconnected lists and consequently different test sequences. The choice between the two is driven by the test cost, since the potential solutions with lower fault coverage are not even considered.

4. Experimental Results

In this section we report some experimental results obtained using the proposed technique on a sub-set of the ITC16 benchmark networks to show its effectiveness when compared to the previous approach [13] and the sub-optimal approach based on the depth-first algorithm [8].

The main reason why not all networks from the benchmark set have been considered is that they contain some constructs that are currently not supported by our tool. The networks from the evaluation set differ in the number and type of reconfigurable modules and therefore in the number of configuration bits, hierarchical depth etc. All these ITC16 benchmark network characteristics are reported in Table 3. For each network in column 1 (*Network*), the columns 2 and 3, give the total number of reconfigurable modules - number of SIBs and ScanMuxes, respectively. The column *Conf. bits* represents the total number of bits that can be used to program all the modules. The *Max depth* column refers to the maximum hierarchical depth of the network (for SIB-based networks this value equals to the maximum number of nested SIBs, according to [10]). The sixth column, labelled *Max path*, indicates the length of the scan path with the highest possible number of scan cells, i.e., flip-flops on it. Finally, the last column *Scan cells* gives the total number of bits present in all segments of the network.

The whole framework setup consists of three modules. First, the evolutionary engine μGP [14] which generates new individuals by applying genetic operators. A next generation of individuals is created based on the fitness values of the newly created offsprings. The second module, the evaluator, is written in Java and works independently. Its role is to provide the complete set of transitions for each of the individuals calling the *Transition* function. Additionally, for each of the individuals generated by the μGP the fitness scores are formed based on the values returned by

Table 3. Benchmark networks list

Network	SIB	SM	Conf. bits	Max depth	Max path	Scan cells
Mingle	10	3	13	4	171	270
TreeBalanced	43	3	48	7	5,219	5,581
TreeFlat_Ex	57	3	62	5	5,100	5,195
TreeUnbalanced	28	-	28	11	42,630	42,630
a586710	-	32	32	4	42,381	42,410
p22810	270	-	270	2	30,356	30,356
p34392	-	96	96	4	27,899	27,990
q12710	27	-	27	2	26,185	26,185
t512505	159	-	159	2	77,005	77,005
N132D4	39	40	79	5	2,555	2,991
N17D3	7	8	15	4	372	462
N32D6	13	10	23	4	84,039	96,158
N73D14	29	17	46	12	190,526	218,869
NE600P150	207	194	401	78	23,423	28,250
NE1200P430	381	430	811	127	88,471	108,148

the *Evaluation* function that calculates the fault coverage and the number of clock cycles required to apply the generated test sequence. The tool is able to read a file containing the network description in various formats, including the ICL. Finally, a separate tool is developed in Java and can be optionally used to further reduce the test cost by manipulating the best individual created by the μGP.

The experiments were run on a server equipped with a dual Intel Xeon CPU E5-2680 v3 and 256 GB of RAM (evolutionary phase) and on a laptop with dual Intel i5-7200U CPU and 8GB of RAM (post-processing phase). The server was used to run the evolutionary engine and perform evaluations for each of the individuals, while the laptop was used to perform the post-processing. To emphasize, the reason behind running the evolutionary and post-processing algorithms on two different platforms is not necessity, but commodity, since the algorithms have been developed in different environments. Additionally, the post-processing phase does not require large amount of RAM so there was no obvious advantage of running this task on the server as well. However, if this phase is executed on the server, wall-clock time would be conservatively reduced up to five times.

For each benchmark RSN, the sub-optimal approach based on the depth-first algorithm that traverses the RSN isomorphic graph structure has been executed (it requires a single run). The depth-first approach is very efficient in terms of time and requires few seconds to complete. The evolutionary approach has also been run on each benchmark and compared with the depth-first approach. The experiments executed on the server have been parallelized using up to 8 cores.

The μGP parameters were configured as follows: n_p set to 200, n_o set to 120, while a steady state of 500 generations was chosen. Concerning genetic operators, the following mutation operators have been enabled: insertion, removal, replacement, alteration, swap; and for crossover: one-point precise/imprecise, two-point precise/imprecise, inver-over [15].

The initial population is composed of:

- individuals that may contain random configurations (due to *random* macro);
- apart from random configurations, individuals may contain partially pre-defined configurations, i.e., in this case configurations generated by the ATPG approach
- an individual with a sub-optimal solution (depth-first) that has been directly inserted into the population - seeding.

Table 4 provides experimental results and is organized as follows: the *evolutionary* segment including columns 2-8 reports results regarding the evolutionary stage, while the second, *post-processing* segment (columns 9-12) provides results obtained by employing described post-processing techniques. For each of the benchmark networks, the wall-clock time (in hours) required by the evolutionary algorithm to reach the steady state is given in column 2 (*Wall-clock time*). The column 3 (*#macros*) reports the total number of macros defined in a constraint file for each of the networks. The number of evaluated individuals and the number of generations used by the evolutionary algorithm are given in columns 4 (*Eval.ind.*) and 5 (*Gen.*),

respectively. Then, the number of configuration (*#conf*) and test (*#test*) vectors as well as the total time in clock cycles (*Test time*) required to apply the test sequence delivered by the evolutionary algorithm are reported in columns 6-8. The wall-clock time for post-processing to be applied is reported in column 9 of the same table. After running post-processing algorithm on the test sequence generated by the evolutionary engine, a potentially modified sequence is obtained for which the number of configuration (*#conf*) and test (*#test*) vectors are given in columns 10 and 11, respectively. The total time (number of clock cycles) needed to apply the aforementioned sequence is contained in column 12 (*Test time*).

A comparison between the presented approach and the two previously described approaches (the evolutionary approach [13] and the depth-first approach [8]) is given in Table 5. For all of the three approaches the table reports the number of configuration vectors (*#cv*) and the number of test vectors (*#tv*), as well as the total time in clock cycles required to apply the generated sequence (*Test time*). In addition, the results obtained resorting to the proposed approach have been confronted with the results from [8] and [13]. The numbers are given in percentages in the last two columns, respectively; they are calculated based on how much is the new *Test time* reduced with respect to the previous results. The same data regarding the comparison is represented visually in the form of a chart in Fig. 9.

Applying the generated test sequences results in achieving full test coverage, i.e., 100%, given the adopted fault model. Furthermore, by only rewriting the *Transition* function which is used to generate the configuration vectors between two test steps we were able to achieve up to 27% decrease in total test cost for 6 out of 16 benchmark networks when compared to the depth-first approach. In some cases, due to the size and complexity of the networks, seeding the population with the sub-optimal solution individual has led the evolutionary algorithm to saturate the population, thus not improving the inserted sub-optimal solution. However, introducing the described post-processing methods led to a further decrease of the total test cost for the remaining circuits, i.e., in total in 14 out of 16 cases. The post-processing has shown to be highly effective even for the two large networks (NE1200P430 and NE600P150). The results for the networks with low hierarchical depth have not been particularly influenced (small improvement or none) by the new technique, probably due to their low hierarchical depth and small number of test vectors. In these cases, the depth-first approach has most likely produced the solution close or equal to the global optimum. Additionally, here we report only basic statistical qualifiers such as minimum, maximum and median values of time reduction due to the limited and insufficient number of benchmark networks. When the proposed approach is confronted to [8], the latter values are 0%, 27% and 10.1%, respectively; when compared to [13], 0%, 26.6% and 7.1% values are derived.

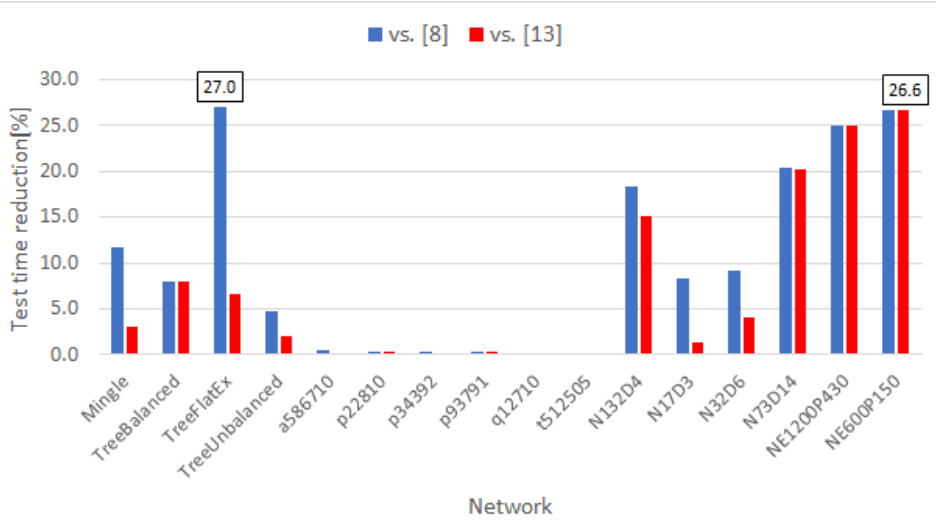

Fig. 9. Reduction chart [%]

5. Conclusions

The present article addresses the issue of minimizing the time required to test the reconfigurable modules in IEEE 1687 RSNs. The proposed methodology is primarily based on evolutionary computation. Additionally, the problem of finding suitable test configurations has been converted into a circuit suitable for applying the automatic test pattern generation procedure. An optimized transition function and some techniques for post-processing the solution delivered by the evolutionary engine have also been presented. Experimental results on the standard set of benchmark networks show the effectiveness of the proposed approach, since the test time has been reduced up to 27% in 14 out of 16 cases, particularly impacting the test time for large networks.

Acknowledgements

The work has been partially supported by the European Commission through the Horizon 2020 RESCUE-ITN project under the agreement No. 722325. The authors want to thank Luigi San Paolo for the environmental setup and the preliminary experiments.

Table 4. Experimental results on the ITC'16 benchmark networks

	evolutionary							post-processing			
Network	Wall-clock time [h]	#macros	Eval. ind.	Gen.	#conf	#test	Test time [cc]	Wall-clock time [min]	#conf	#test	Test time [cc]
Mingle	8	13	49,169	576	6	7	2,135	<1	6	7	2,014
Tree Balanced	6	47	43,914	500	7	8	69,369	<1	7	8	63,843
TreeFlat_Ex	13	38	34,931	1,178	22	6	52,086	<1	22	6	52,086
TreeUnbalanced	5	31	31,329	818	17	12	1,026,333	<1	12	12	1,021,023
a586710	15	14	49,129	500	5	5	299,624	<1	5	5	298,210
p22810	32	78	21,001	500	2	3	152,937	1	2	3	152,399
p34392	68	32	26,292	1,069	5	5	196,223	<1	5	5	196,128
p93791	27	48	29,932	500	4	5	708,878	1	4	5	706,242
q12710	24	16	19,900	500	2	3	131,022	<1	2	3	131,022
t512505	8	40	21,279	500	2	3	386,024	<1	2	3	385,440
N132D4	3	46	47,177	552	5	6	38,731	<1	5	6	31,645
N17D3	7	15	59,384	509	4	5	3,841	<1	4	5	3,797
N32D6	3	15	36,786	419	4	5	904,974	<1	4	5	856,406
N73D14	2	36	34,075	774	14	13	6,078,868	<1	13	13	4,762,150
NE1200P430	78	317	48,902	500	127	128	21,515,705	4k	127	128	16,131,171
NE600P150	19	286	45,857	500	78	79	3,726,726	180	78	79	2,735,016

Table 5. Comparison of the experimental results with the approaches from [8] and [13]

Network	Depth-first [8]			Evolutionary [13]			Proposed approach			Comparison	
	$\#cv$	$\#tv$	Test time [cc]	$\#cv$	$\#tv$	Test time [cc]	$\#cv$	$\#tv$	Test time [cc]	Test time reduction vs. [8]	Test time reduction vs. [13]
Mingle	6	7	2,282	6	7	2,078	6	7	2,014	11.7%	3.1%
Tree Balanced	7	10	69,369	7	8	69,369	7	8	63,843	8.0%	8.0%
TreeFlat_Ex	5	6	71,341	22	6	55,776	16	6	52,086	27.0%	6.6%
TreeUnbalanced	11	12	1,071,799	12	12	1,042,450	17	12	1,021,023	4.7%	2.1%
a586710	4	5	299,624	5	5	298,241	5	5	298,210	0.5%	0.0%
p22810	2	3	152,937	2	3	152,937	2	3	152,399	0.4%	0.4%
p34392	4	5	196,702	5	5	196,505	5	5	196,128	0.3%	0.2%
p93791	4	5	708,878	4	5	708,878	4	5	706,242	0.4%	0.4%
q12710	2	3	131,022	2	3	131,022	2	3	131,022	0.0%	0.0%
t512505	2	3	386,024	2	3	386,024	2	3	385,440	0.2%	0.2%
N132D4	5	6	38,731	5	6	37,257	5	6	31,645	18.3%	15.1%
N17D3	4	5	4,143	4	5	3,851	4	5	3,797	8.4%	1.4%
N32D6	4	5	942,470	4	5	893,017	6	5	856,406	9.1%	4.1%
N73D14	12	13	5,978,047	13	13	5,967,137	13	13	4,762,150	20.3%	20.2%
NE1200P430	127	128	21,515,705	127	128	21,515,705	128	128	16,131,171	25.0%	25.0%
NE600P150	78	79	3,726,726	78	79	3,726,726	78	79	2,735,016	26.6%	26.6%

References

1. IEEE standard for access and control of instrumentation embedded within a semiconductor device, *IEEE Std 1687-2014* (Dec 2014) 1–283.
2. IEEE standard for test access port and boundary-scan architecture, *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)* (May 2013) 1–444.
3. F. G. Zadegan, U. Ingelsson, G. Carlsson and E. Larsson, Design automation for IEEE p1687, in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, IEEE2011, pp. 1–6.
4. K.-J. Lee and M. A. Breuer, A universal test sequence for cmos scan registers, in *Proceedings of the IEEE 1990 Custom Integrated Circuits Conference*, 1990, pp. 28–5.
5. S. Maka and E. J. McCluskey, Atpg for scan chain latches and flip-flops, in *1997 15th IEEE VLSI Test Symposium (VTS)*, 1997, pp. 364–369.
6. F. Yang, S. Chakravarty, N. Devta-Prasanna, S. M. Reddy and I. Pomeranz, On the detectability of scan chain internal faults an industrial case study, in *2008 26th IEEE VLSI Test Symposium (VTS)*, 2008, pp. 79–84.
7. A. T. Dahbura, M. U. Uyar and C. W. Yau, An optimal test sequence for the JTAG/IEEE P1149.1 test access port controller, in *IEEE International Test Conference, 1989. Proceedings. Meeting the Tests of Time.*, 1989, pp. 55–62.
8. R. Cantoro, M. Montazeri, M. Sonza Reorda, F. G. Zadegan and E. Larsson, On the testability of IEEE 1687 networks, in *2015 IEEE 24th Asian Test Symposium (ATS)*, 2015, pp. 211–216.
9. R. Cantoro, M. Palena, P. Pasini and M. Sonza Reorda, Test time minimization in reconfigurable scan networks, in *2016 IEEE 25th Asian Test Symposium (ATS)*, 2016, pp. 119–124.
10. A. Tšertov, A. Jutman, S. Devadze, M. Sonza Reorda, E. Larsson, F. G. Zadegan, R. Cantoro, M. Montazeri and R. Krenz-Baath, A suite of IEEE 1687 benchmark networks, in *2016 IEEE International Test Conference (ITC)*, 2016, pp. 1–10.
11. D. Ull, M. Kochte and H. J. Wunderlich, Structure-oriented test of reconfigurable scan networks, in *2017 IEEE 26th Asian Test Symposium (ATS)*, Nov 2017, pp. 127–132.
12. A. Eiben and J. Smith, *Introduction to Evolutionary Computing* (Springer Berlin Heidelberg, 2015).
13. R. Cantoro, L. San Paolo, M. Sonza Reorda and G. Squillero, An evolutionary technique for reducing the duration of reconfigurable scan network test, in *2018 IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, April 2018, pp. 129–134.
14. E. Sanchez, M. Schillaci and G. Squillero, *Evolutionary Optimization: the μGP toolkit* (Springer Science & Business Media, 2011).
15. Genetic operators https://sourceforge.net/p/ugp3/wiki/Genetic%20operators/.